Problem 10 Editorial: Steve was obliterated by a sonically-charged shriek
8 Points

Problem ID: `warden`
Rank: `3`

Overview

Problem statement: https://github.com/calico-team/calico-fa22/[TODO replace]
Interactive tool: https://www.desmos.com/calculator/smz06e84gt

For our very first interactive problem in a CALICO contest, we tried making something fun and accessible with a lot of valid approaches! The key observation between all of them is that although the output of each pulse query contains redundant information, they become much easier to work with once simplified into just the distance between the pulse location and Steve. After that, we have lots of ideas we can build upon such as divide and conquer, iterative optimization, or just plain old geometry to hunt Steve down.

This problem idea was conceived in the early snapshots of Minecraft 1.19 before the mechanics of the warden were finalized, so the method described in this problem is based on our *idea* of how the warden finds its prey, rather than how it actually does in the game!

Main Test Set

There are many approaches that solve this problem, but we'll outline four notable distinct ideas. They all use a common method of post-processing the query output.

To start, observe the path of the distance we're given after every pulse query:

$$\text{Warden} \Rightarrow \text{Pulse Location} \Rightarrow \text{Steve} \Rightarrow \text{Pulse Location} \Rightarrow \text{Warden}$$

Notice that it's symmetric before and after Steve! Since the second half of the path doesn't provide any new useful information for us, we can simplify the query output without losing information by dividing by 2 to get the one-way path distance.

The quantity that we're actually interested in is the distance from the pulse location to Steve. We don't really care about the distance between the Warden and the Pulse Location, as that's already known to us. To simplify further, we can subtract this quantity from the one-way path distance to get just the distance between the pulse location and Steve.

The following formula gives the distance between the pulse location and Steve for each pulse:

$$d(x_p, y_p, \mathbf{X_S}, \mathbf{Y_S}) = p(x_p, y_p) \,/\, 2 - \mathrm{d}(0, 0, x_p, y_p)$$

where d is the Euclidean distance function, p is the pulse query output function, (x_p, y_p) is the pulse location, and $(\mathbf{X_S}, \mathbf{Y_S})$ is Steve's location.

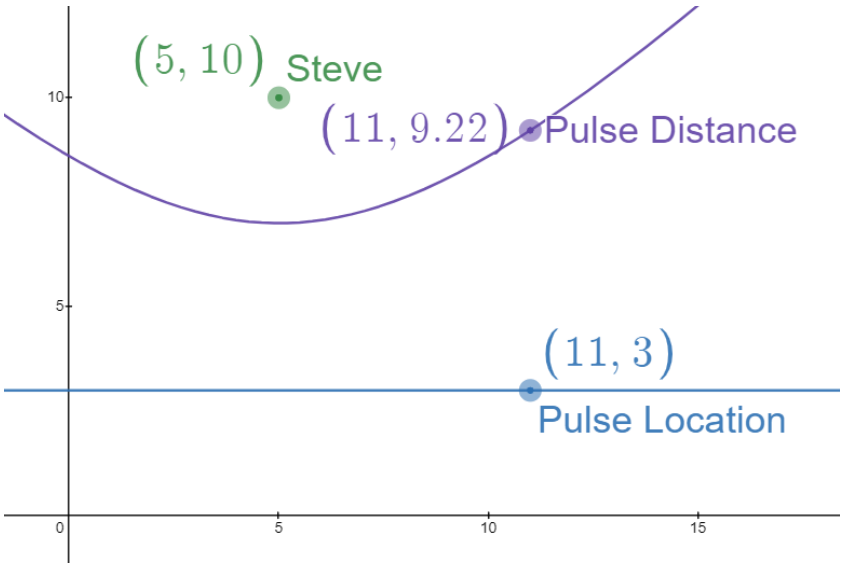
From here, we have a couple of different approaches we can take to find Steve.

Valley Finding

Searching for Steve in 2D space directly is a challenge, but if we can somehow separate the search for each coordinate independently, the problem becomes much easier.

Suppose then that we fix our search to some arbitrary horizontal line for now, and we're interested in finding $\mathbf{X_S}$, the horizontal coordinate of Steve. The shortest distance between this line and Steve must be the point on the line that's directly above or below Steve. This is because if we move left or right from this point, we'll get further away, increasing the distance. In other words, this minimal distance point has an x-coordinate of $\mathbf{X_S}$!

Thus, we can find $\mathbf{X_S}$ by finding the x-value along any horizontal line such that the x-value minimizes the distance between the pulse location and Steve. Here's an interactive tool to see what this is like.



Since the distance increases the further away we stray from Steve's true x-coordinate, we can find the value of Steve's x-coordinate by performing a ternary search on the x-axis to find the value that minimizes pulse distance. Now that we know the x-coordinate, we know that Steve's y-coordinate must be either above or below the last pulse location by a distance of the last pulse distance. We can figure out which case this is by performing one more query. Alternatively, we can also just perform a second ternary search along the y-axis to find Steve's y-coordinate as well.

With both Steve's x and y coordinates, we can now blast him into smithereens!

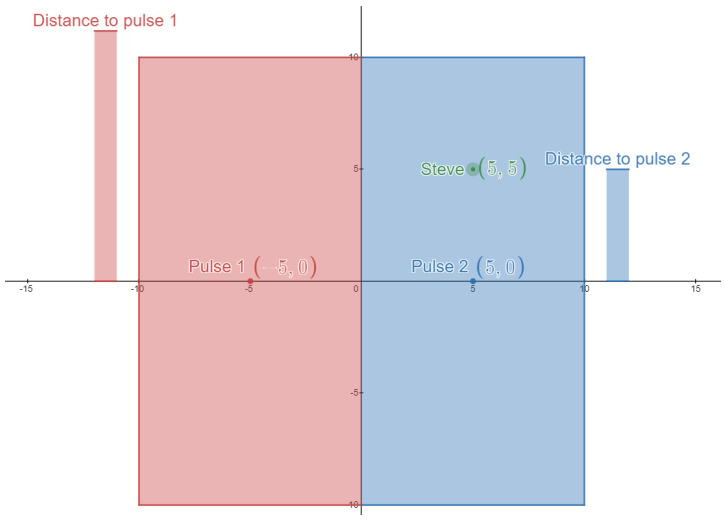
© 2022 California Informatics Competition

Binary Space Partitioning

This approach works by cutting the space that Steve could be in by half each step. In the beginning, Steve can be anywhere in the square with radius 10^5 centered in the origin. However, we can partition this potential space by cleverly choosing pulse locations. By repeating this process, we can eventually narrow down the region that Steve could lie in to well within the bounds of our error tolerance.

We begin by splitting the region that Steve could be in into two parts and then sending a pulse in the center of each region. Whichever pulse is closer determines which region Steve is hiding in. This general technique is called binary space partitioning and is commonly used in algorithms for computer graphics.

For example, say we know Steve may be anywhere in the square with corners (-10, -10), (-10, 10), (10, -10), (10, 10). We know that Steve must be in either the left side or right side of this square, so we can partition this space into two rectangles. Let's call the left side A with corners (-10, -10), (-10, 10), (0, 10), (0, -10) and the right side B with corners (0, -10), (0, 10), (10, 10), (10, -10). We can then send a pulse to (-5, 0) in A and (5, 0) in B. If the pulse distance from (-5, 0) must be smaller than the pulse distance from (5, 0), Steve must be in A, and vice versa for B. Here's an interactive tool to see what this is like.



In terms of bounds, each iteration requires 2 queries. Thus, we can repeat this process 75 times. Each iteration halves the potential space. We initially start with a potential space with area $(2 \times 10^5) \times (2 \times 10^5) = 4 \times 10^{10}$, but after 75 iterations, we cut down the space into a mere $4 \times 10^{10} \times (1/2)^{75} \approx 1.059 \times 10^{-12} \approx 0.000000000001058$, which is well within our error bound.

© 2022 California Informatics Competition

Iterative Optimization

Another idea we can take is that we can start by taking a guess at Steve's location, then slowly adjust our guess to get closer and closer to where Steve actually is until we're within the error bounds. The key observation is that we're essentially trying to optimize to find the minimum of the pulse distance function. This technique is called coordinate descent, and is often used in optimization algorithms.

To implement this approach, we can start by guessing Steve's location at (0, 0), then query nearby points such as (1, 0) and (0, 1) to measure the change in pulse distance along each direction. Depending on which axis has a greater decrease and the magnitude of the decrease, we can adjust our guess accordingly. There's some flexibility in the exact details for how this is done, and it'll likely take some adjusting to get a good learning rate.

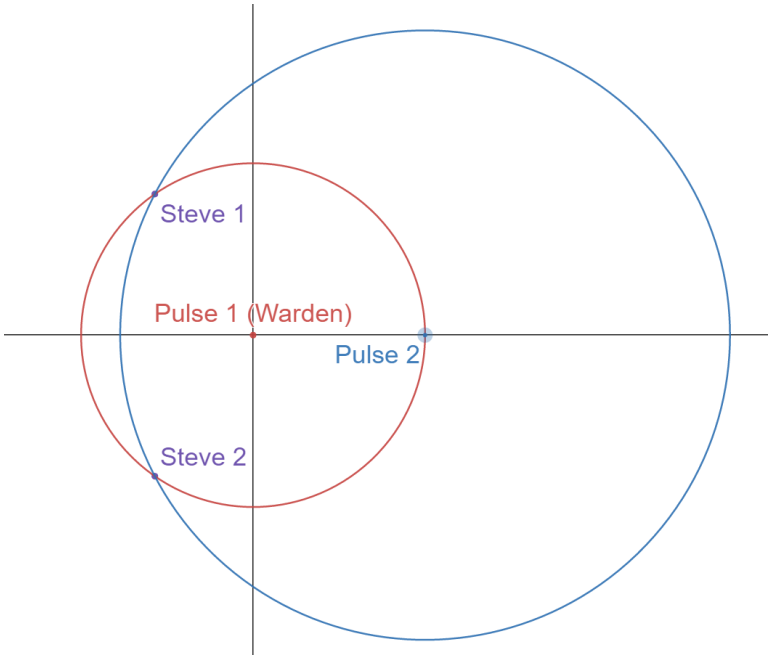
Although coordinate descent is a bit overkill and is usually used to solve more complex problems in machine learning, it serves as a good starting point in tackling the harder version of this problem, warden2.

© 2022 California Informatics Competition

Geometry

Finally, there are some geometrical solutions that can find Steve in only 3 pulses! There are many ways to achieve this, but here is one such solution.

By sending the first pulse at (0,0) the distance between steve and the origin can be found. Call this distance *r*. We now know that Steve must be somewhere on the circle of radius *r* that is centered around the origin. Next send a pulse at (*r*, 0); this gives a second circle that narrows down Steve's location to just two potential spots (see the diagram below). Try it for yourself with this interactive graph.



Note that the points of pulse 1, pulse 2 and steve 1 make an isosceles triangle. Thus, we can apply the law of cosines. Let r be the result of pulse 1 and d be the result of pulse 2. Then $d^2 = 2r^2 - 2r^2 cos(\theta)$, meaning that $cos(\theta) = (2r^2 - d^2)/(2r^2)$. From this we can find θ and also the two potential points where Steve could be. Now we simply send a pulse at one of the two potential locations. If it is within our error bound, then we can blast that location to pass the test case. Otherwise, we simply blast the other potential location.

Another geometrical solution that requires only three pulses is given by this awesome Math StackExchange post by calculating centroids. Notably, it always queries the same three points, is super easy to implement, and is robust to scenarios where rounding errors may cause the circles to not intersect perfectly!

© 2022 California Informatics Competition